EDF Scheduling of Real-Time tasks on multiple cores

OBI NNAMDI ELIJAH

***Abstract*—As computer technologies have evolved in pursuit of better performance, so also has the complexity of software computations greatly increased. For example, real-time schedulers for multi-core platforms are known to be subject to the so-called scheduling anomalies, where increasing the number of CPU or the CPU frequency sometimes turns a system non-schedulable causing deadline miss. In the context of our research topic, particular focus is given to the challenge of designing efficient and real-time task schedulers for multiple core platforms, which is well known to be more cumbersome than for Uniprocessor based systems.**

This paper presents an overview discussion into the scheduling technique of real-time tasks on multiple cores using EDF (Earliest Deadline First) scheduling. A further look into different algorithms for scheduling tasks based on EDF methodology would be looked at and at the end, a final conclusion would be drawn for the best approach after comparison.

***Index Terms*—Multi-Core Real-time Scheduling, Real-Time Operating Systems, EDF(Earliest Deadline First)**

I. INTRODUCTION

The use of multi-core and multi-processing computing has increased exponentially across a wide range of application fields, from embedded systems to personal computers. This is an ongoing research subject, as seen by the various approaches of scheduling real-time jobs on multi-core computers utilizing EDF scheduling. As far back as in the 1960s, it was observed by Liu, the inventor of RM and EDF, that: *“Few of the results obtained for a single processor generalize directly to the multiple processor case; bringing in additional processors adds a new dimension to the scheduling problem. The simple fact that a task can use only one processor even when several processors are free at the same time adds a surprising amount of difficulty to the scheduling of multiple processors”.*

As discussed in several quarters, ”Per chip core counts are projected to rise dramatically as technological innovation continues”. This is supported by trends from key chip makers. For example, Intel in the past announced plans for a core architecture with more than 50 cores per chip; now, we have big complicated systems. AMD’s ”Magny-Cours” Opteron processors, with 12 cores per chip, and Intel’s ”Beckton” Xeon processors, with eight multi-threaded cores per chip, are just two examples.A processor is unquestionably an essential component of a computer. A processor is capable of executing sets of instructions, with a single instruction referred to as an output in one case and an input in another. Thus, it is feasible to design a set of instructions that are arranged logically and execute certain types of calculation; this is known as a computer program.

A. HISTORY

A short overview of the history of computational systems is a necessary step in understanding the overall set-up of this paper. Below we will briefly look at the topic of single core and multiprocessor systems.

1) single core processors: It is important to introduce some key hardware issues by briefly reviewing single-core processor systems. As a result, we begin by providing a general overview of a single-core processor platform (see Figure 1).

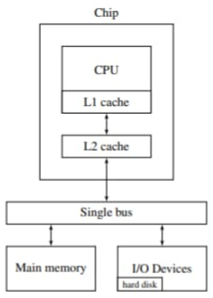


Fig. 1. Illustration of a computer according to the von Neumann architecture [2]

Von Neumann’s architecture is the foundation of many computer systems. A computational system, according to this design, consists of a CPU (Central Processing Unit), memory, and Input/Output (I/O) devices coupled via a single bus. The CPU is the component in charge of application execution. Data and instructions are both stored in memory and make up an application. As a result, a CPU running an application must get information (data and instructions) from main memory. As is obvious, the program execution speed is determined not only by the CPU’s processing capability, but also based on the time spent to access the system memory.From the invention of CPU in the early 1970s to the present day, the progress of CPU has been phenomenal; there is likely no other similar example in the technical world. [2]

CPU speed improvements, Instruction-Level Parallelism (ILP) methods, pipelines, and memory caches have all been used to improve performance (as measured by the amount of time it takes to complete a particular operation). The time of a clock cycle reduces as CPU speed increases (in one clock cycle, a CPU can perform a basic operation such as, for instance, fetching an instruction). Simultaneous Multi-threading is one of the ILP strategies used to improve a CPU’s overall efficiency (SMT).SMT transforms a single physical CPU system to look like one of multiple CPUs. A physical CPU has many logical CPUs for this reason. Each logical

CPU is made up of a nearly complete set of CPU hardware, including registers, pipelines, and other components, but they all share the execution unit. This allows several threads (or tasks) to be kept ready to run at the hardware level, lowering the context switch (or task switch) cost; that is, the cost of switching the execution unit from one task to another. A SMT system is made up of several CPU from the standpoint of the operating system. In practice, however, only one task may be completed at a time. As a result, an SMT system is essentially a single-CPU system.An implementation of this is found in the Intel hyper-threading. [2]

The primary memory accesses are too sluggish in comparison to the CPU performance, causing the CPU to stall while waiting for the information to be returned. The primary function of caches is to minimize memory access latency. A cache is a smaller, quicker memory that keeps copies of the application's data/instructions that the CPU has recently utilized. Caches are often structured in a hierarchy of bigger and slower caches, as seen below. When the CPU wants to read from or write to a place in main memory, it first examines the level 1 (L1) cache; if it is successful (i.e. the information is stored in L1 cache), the CPU continues.If the L1 cache (the fastest in terms of access) misses (if the information is not saved in L1 cache), the next bigger cache (L2) (slower than L1) is tested, and so on, before accessing the main memory. When the necessary information is not stored in either of the previously described memory units (caches and main memory), the CPU must get it from an I/O device, such as a hard drive, which takes orders of magnitude longer to access than even main memory. [2]

2) multiprocessors systems: CPU have become smaller, quicker, and more computationally capable with each iteration over the previous few decades. However, with each iteration, heat and power consumption has increased, forcing CPU makers to switch from single-core to multi-core processing designs in order to solve this. There was no terminological distinction between CPU and processor prior to the multi-core processor design, which incorporates numerous CPU on a single chip (see Figure 2).

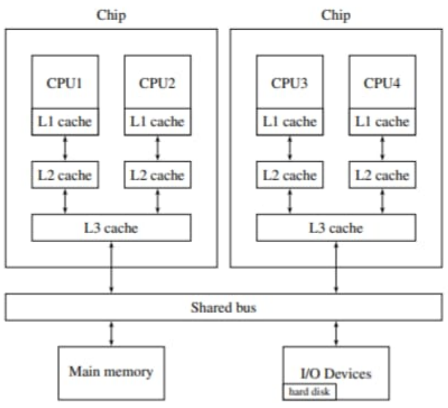


Fig. 2. Illustration of a multiprocessor architecture according to the shared memory model

[2]

In general, the word processor was used in literature to refer to a single-core system, as opposed to the term multiprocessor, which was used to refer to systems having two or more CPU.When referring to more than one CPU (whether or not they are on the same chip), we use the phrase multiprocessor, and when referring to only one CPU, we use the term processor. When referring to a system with only one CPU, the term uniprocessor is used.

It is important to note that multiprocessor architectures are not a novel notion, and there is no single standard design for multiprocessor systems. A multiprocessor is categorised based on the types of processors that comprise it. A multiprocessor can be described as identical, heterogeneous, or uniform. Identical means that all processors have the same properties, such as clock speed, architecture, cache capacity, and so on; heterogeneous means that they have varied capabilities and speeds.Finally, with uniform multiprocessors, one processor may be y times faster than another, which means it performs all conceivable jobs y times quicker. Multiprocessor systems are classed as having shared or distributed memory, based on their memory model. All processors in a shared memory paradigm (see Figure 2) share the main memory. The multi-core architecture is one typical implementation of the shared memory model, and it is currently the new path that manufacturers are focused on [2].

Communication between CPU, caches, memory, and I/O devices is an essential performance problem in a multi-core design.Initially, such communication was accomplished via a shared bus (see Figure 2). However, because of the shared bus's significant latency and congestion, chip makers began inventing innovative communication techniques. To name a few, these techniques are AMD HyperTransport, Intel Quickpath, and Tilera iMesh. In the shared memory paradigm, access times to main memory are consistent from any processor, regardless of the technology used to connect it other processors. This is referred to as Uniform Memory Access (UMA).

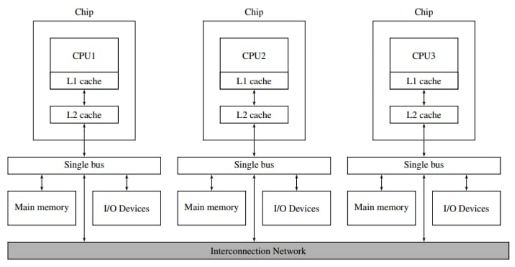


Fig. 3. Illustration of a multiprocessor architecture according to the distributed memory model

[2]

B. PROBLEM DESCRIPTION

The design of efficient real-time CPU schedulers for multi-core and multi-processor platforms, has been greatly recognized to be more complex than it is for single-CPU systems. Real-time schedulers for multi-core systems, for example, are known to be vulnerable to so-called scheduling anomalies, in which increasing the number of CPUs or the CPU frequency occasionally makes it difficult to schedule a system, resulting in missed deadlines.

However, through research and industry application, we have significantly improved support for multicore/processor platforms for real-time tasks, where the system's correctness is determined not only by its functional correctness, but also by its ability to allow real-time tasks to respect their temporal constraints. We may classify multi-core scheduling approaches into two groups: Tasks in global scheduling can be allocated around cores/CPUs based on the internals of the scheduling strategy; in partitioned scheduling, tasks are statically partitioned (that is, priorities are assigned offline, once and for all) among the available processors, thereafter with a single-processor scheduler used on each CPU. They are both relevant and viable solutions in today's operating systems. When it comes to multi-processors, for example, the Linux kernel supports several real-time scheduling algorithms based on fixed-priority or deadline-based scheduling. However, as we would see in the course of this paper, there are certain constraints that needs to be thought through and solved in order to find an optimal scheduling for tasks on multi-cores using EDF scheduling.

C. PAPER ORGANISATION

This paper is organised as follows: After a review of the relevant research topics in Section 2, I endeavoured to provide additional background and definitions in Section 3 about fundamental concepts that were used in most papers covering this topic. In Section 4, I described the approach of adaptive partitioned scheduling as understood and lastly, in Section 5 conclusions were drawn.

II. RELATED WORKS

Other research on multi-processor real-time scheduling have focused on either hard real-time systems (where all task deadlines must be met) or soft real-time systems (where a given number of missed deadlines can be permitted). In the case of hard real-time systems, research indicates that global EDF may be tweaked to have an optimal utilization constraint for fixed-job-priority algorithms and that optimal multiprocessor scheduling algorithms (based on global scheduling) exist. All of these methods, however, are rarely employed in reality, with most operating systems focusing on partitioned or global fixed-priority or EDF scheduling.Based on the above, partitioned scheduling is generally preferred in hard real-time systems (where execution times are more stable and transient overloads are less likely to occur, and meeting all deadlines is a critical factor), whereas global scheduling (particularly global EDF) is more commonly used in soft real-time systems (where the tardiness guarantees provided by global EDF are generally enough, but execution times are less predictable and transient overloads are more likely to happen).

Previous work conducted empirical investigations evaluating the benefits and drawbacks of global versus partitioned scheduling in various settings. Previous research on hard real-time systems has largely focused on optimum off-line partitioning, such as that achieved by integer linear programming approaches.Other studies, on the other hand, focused on more dynamic real-time systems that necessitate online partitioning techniques. Various writers evaluated the usefulness of bin-packing heuristics such as first-fit, worst-fit, and next-fit in this setting, which have been extensively studied in other situations such as memory management (see for example the seminal works by Graham and Johnson, or more recent works and comprehensive surveys on the topic as found within the references cited on this work [3]. Many of these studies concentrate on the absolute approximation ratio, or the smallest number of bins required to pack a number of objects of varying weights while employing the aforementioned simple bin-packing heuristics, as opposed to the optimum number that would have sufficed using an ideal technique. Some researchers concentrated on the asymptotic value of such an approximation ratio as the size of the issue increases to infinity. For example, the 12/7 roughly 1.7143 bound for the first-fit heuristic is an intriguing discovery in the domain. However, because many of these studies are not concerned with the scheduling of real-time tasks, they do not investigate the usefulness of the aforementioned heuristics on the performance attained when scheduling various real-time task sets in terms of slack and/or tardiness. [4]

Clustered scheduling has been proposed as a compromise to address the constraints of partitioned and global algorithms. This method takes use of the clustering of cores around multiple layers of shared caches.The system is divided into clusters of cores that share a cache, and jobs are statically allocated to clusters (as in partitioning) but globally scheduled inside each cluster. . Many unsolved problems arise when clustered algorithms are implemented on real-world systems. What is the appropriate level of shared cache to utilize for clustering? Will the cluster size selected function equally well for HRT and SRT systems? How do various preemption- and migration-related overheads compare to scheduling overheads? [4] In the end, it can be said that Clustered schedulers exist somewhere in the middle of Partitioned and Global scheduling, where available processors are partitioned into clusters to which jobs are statically allocated, yet tasks are globally scheduled in each cluster. The employment of partitioned or clustered scheduling strategies in a multi-core system introduces the additional difficulty of how to split jobs among cores or clusters of cores.

III. DEFINITION AND BACKGROUND

The system under consideration consists of a set $\gamma = \{\tau_i\}$ of real-time jobs/tasks τ_i, that will be scheduled on a platform made up of one or more CPU with a sum of M similar cores. Each real-time task τ_i may be represented as a stream of jobs $\{J_{i,k}\}$, where each job $J_{i,k}$ arrives (becomes available for execution) at time $r_{i,k}$ and terminates at time $f_{i,k}$ after running for time $c_{i,k}$ ($f_{i,k}$ can be seen to be dependent on the scheduler). Furthermore, each task has a relative deadline Di, and each job $J_{i,k}$ must be completed within its absolute deadline of $d_{i,k} = r_{i,k} + D_i$. If $f_{i,k}$ is less than or equal to the job's absolute deadline $d_{i,k}$, the job met its deadline; otherwise, the deadline was missed. Task τ_i meets all of its deadlines if $\forall k, f_{i,k} \leq d_{i,k}$; Because $d_{i,k} = r_{i,k} + D_i$, this condition is frequently stated as $\forall k, f_{i,k} - r_{i,k} \leq D_i$.

Thus, Job $J_{i,k}$ tardiness is defined as $\max\{0,\ f_{i,k} - d_{i,k}\}$. Inter-processor migration has always been limited in real-time systems. The expense of context switching from one processor to another is banned in many systems. Recent tests have demonstrated that scheduling algorithms that support inter-processor migration outperform scheduling techniques that do not support migration. According to the degree of migration, scheduling algorithms fall into one of three types, namely:

No Migration
Task Level Migration
Job Level Migration

Usually, there are typically three broad approaches used when scheduling on multi-cores/multi-processors, namely:

Partitioned scheduling
Global scheduling
Hybrid scheduling

A. Global scheduling

All ready jobs are placed in a global priority queue in this manner, and the scheduler picks the highest priority task from the queue for execution. Even after preemption, a task can be assigned to any available processor/CPU. In contrast to partitioned scheduling, a task is not assigned to a specific processor in global scheduling, and task (job) migration is possible. For example, the global EDF (G-EDF) picks the m highest priority (earliest deadline) ready tasks to execute on the multiprocessor system's m processors at time t. For scheduling on multiprocessors/multi-cores, global scheduling algorithms provide the following benefits over partitioned scheduling techniques:

- Preemption are often lower since the scheduler must preempt a job when there is no idle processor yet a task has to be scheduled. Some global scheduling methodologies are work-conserving, which means that no ideal processor will evolve if a ready task has to be scheduled to it
- When a task execution time is less than its worst-case behavior, the time can be shared by all tasks, rather than just a subset of tasks as we would naturally find with partitioned scheduling.
- This strategy is better suited for open systems where load balancing and task allocation are not required for task-set modifications.

For scheduling on multiprocessors, global scheduling algorithms have the following drawbacks over partitioned scheduling algorithms:

- For ready tasks queuing, global scheduling use a single queue. The queue length is considerably long, and accessing the queue can take long time.
- Since a job would regularly be moved from one processor to another, this results in a significant migration overhead for the system.

B. Partitioned scheduling

Tasks are distributed across available processors in this technique. It implies that a task is allotted to a certain processor and is exclusively scheduled on that particular processor alone. As a result, the multiprocessor/multi-core system degrades into a collection of isolated uni-processor systems, with each CPU/core running a uni-processor scheduling algorithm to schedule the tasks allocated to that processor. Unlike with global scheduling, jobs in the partitioned technique are not permitted to transfer from one processor to another. For example, in multiprocessor scheduling using partitioned EDF (P-EDF), a uni-processor scheduling approach called Earliest Deadline First is implemented on each processor individually to schedule the task set.

The task-set must be partitioned into smaller discrete groups of tasks in the Partitioned multi-processor/multi-core scheduling approach. For this partitioning procedure, numerous heuristic approaches are available. As a result, the partitioned technique divides the multiprocessor scheduling issue into two parts: breakdown the task set into m subsets of tasks, so that each subset can be scheduled, where m is the number of processors, with running an independent uni-processor scheduler for each core. EDF is the uni-processor scheduling method in this scenario. There are several polynomial time heuristics available to divide the task set on multi-processor/multi-core systems based on bin-packing techniques that are known to be NP-hard in the strict sense.Three of the heuristics methods for tasks partitioning will be briefly detailed here,with the rest of the algorithms in For scheduling on multiprocessors, partitioned scheduling algorithms provide the following benefits over global scheduling techniques:

- A task's worst-case execution time may be exceeded; this circumstance only impacts tasks on the same processor.
- Since each job operates on the same processor, migration cost penalty is completely eliminated.
- Since a priority queue is maintained for each processor, the queue length is kept to a barest minimum and queue access is faster than with a global priority queue.
- When all tasks have been assigned to processors, the major advantage of employing this strategy for multi-core scheduling is that it reduces multiprocessor/multi-core scheduling to single processor/core scheduling.

The following can be stated as drawbacks of partitioned scheduling algorithms over global scheduling:

- The biggest drawback of this strategy is the assignment of tasks to processors. Finding the optimal task assignment to processors is a bin-packing issue that is NP-Hard in the strictest sense.
- Partitioned approaches do not conserve effort; for example, a CPU may sit idle while certain ready jobs to be scheduled on another processor are missing their deadlines.

The performance of the partitioned scheduling algorithms is determined by the bin packing algorithms used to distribute jobs among the multi-core system's processors. On a multi-core system with m cores, the bin packing techniques cannot guarantee task partitioning that achieves total utilization greater than (m + 1)/2. As a result, in the worst-case situation, the partitioned scheduling methods utilize slightly more than half of the multi-core system's processing capability for task execution, while the remaining nearly half remains idle [3].

C. Hybrid Scheduling

Depending on the hardware architecture, a task in global scheduling may move from one processor to another often, resulting in a very high migration overhead. This job migration causes excessive communication loads and cache misses, increasing the worst-case execution time, which would not occur in the fully partitioned or non-migration scheduling options. Partitioned scheduling, on the other hand, fragments the entire available processing capacity, leaving a considerable percentage of processing capability unused and idle. The greatest usage constraint in the completely partitioned technique is just around half of the entire processing capability, resulting in very low utilization.Furthermore, certain systems can be planned using a completely partitioned or fully global scheduling technique since some tasks are not authorized to migrate while others are allowed.

Cluster-based scheduling, a recently proposed more broad hybrid scheduling technique, may be classified as an extension of partitioned and global scheduling protocols [43]. Tasks are statically assigned to cluster in this technique(partitioning), and then tasks inside a cluster are scheduled globally. Cluster-based scheduling is classified into two types: physical and virtual clusters. Physical cluster-based scheduling assigns processors to each cluster statically, and each cluster contains a subset of processors from the available multi-processor system.

D. Other necessary topics

Before we go any further, it is necessary to discuss about several key topics related to this study.

1) No Migration: Every task is allocated a particular processor that may schedule it. Migration between processes is not permitted in this category, hence the approach is also known as partitioned. If the available number of processors is m, there are m distinct subsets of task sets. A subset of tasks is assigned to a processor and must be scheduled exclusively on that processor.

2) Task Level Migration: A task's jobs may run on separate processors; nevertheless, each job must run wholly on a single processor. As a result, a job's run-time context can only be maintained on one processor, although task-level context migration is feasible. Restricted migration is another name for task level migration.

3) Job Level Migration: Inter-processor movement of task's job is unrestricted. A task's job can be migrated and transferred from one processor to another; however, parallel execution of a job is prohibited, therefore a job cannot be scheduled on more than one processor at the same time. Full migration is another name for task level migration.

4) Next Fit: This approach keeps the processors in order and allocates tasks to each processor. Because many tasks may be allocated to that processor in this manner, the partition can be scheduled using a suitable single core scheduling algorithm. A detailed report on this can be found in [4].

5) First Fit: The next fit approach is accomplished by assuming that a task will be given to the very first available processor to which it matches, as explained in [4].

6) Best Fit: Each work is allocated to a processor that can fit the specified task, and the residual unused processing capacity is reduced. Srinivasan [4] introduced this approach using EDF as a uni-processor scheduling algorithm in section 2.4. The time complexity of the next fit partitioning technique is O(n) times the time complexity of the uni-processor task scheduling analysis, whereas the time complexity of the first fit and best fit partitioning methods is O(mn) times the time complexity of the uni-processor task scheduling analysis. When the task sets are partitioned in P-EDF, a uni-processor dynamic priority scheduling method EDF is used to schedule the task sets allocated to that core.

IV. ADAPTIVE PARTITION SCHEDULING

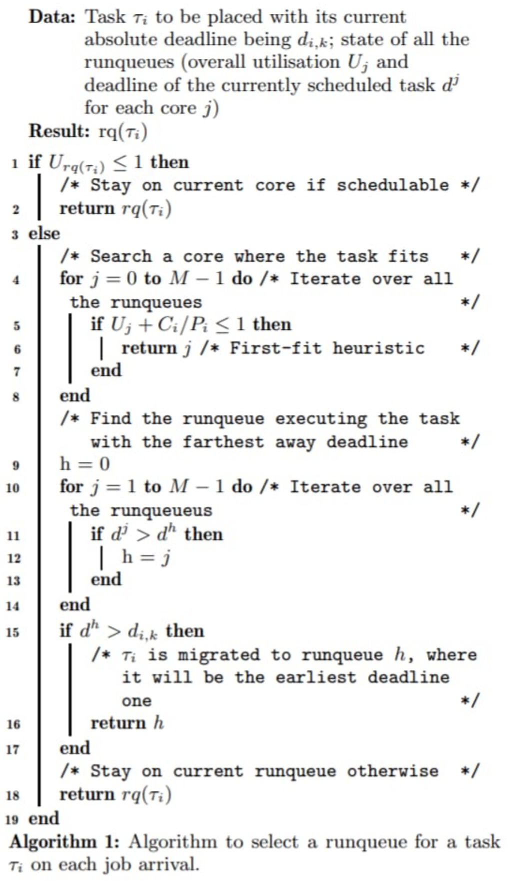
As stated in Abeni and Cucinotta's paper [3]: *The adaptive partitioning migration strategy implements a restricted migration scheduling algorithm based on r-EDF [ref5]. Since all the runqueues are ordered by absolute deadlines (implementing the EDF algorithm, so that $U^{lub} = 1$), the algorithm is named adaptively partitioned EDF (apEDF). In more details, in a scheduling algorithm based on restricted migrations a task τ_i that starts to execute on core/CPU j cannot migrate until its current job is finished (each job $J_{i,k}$ executes on a single core/CPU, and cannot migrate). Hence, using the Linux terminology, the scheduler is based on a "select task runqueue" operation (invoked when a new job arrives), but does not use any "push" nor "pull" operation.*

A. The Basic Algorithm

To clarify the explanation of the apEDF method, let $rq(\tau_i)$ be the run-queue into which τ_i has been entered (that is, the core on which τ_i runs or has ran) and $U_j = \sum_{\{i:rq(\tau_i)=j\}} C_i/P_i$ denote the utilisation of the tasks executing on core j. Furthermore, let d^j represent the absolute deadline $d_{h,l}$ of the work that is now running on core j, or infinity if core j is idle.

When a task τ_i is formed, its runqueue is initially set to $0(rq(\tau_i) = 0)$ and will ultimately be set to an appropriate runqueue when the first job comes (the task wakes up for the first time). When job $J_{i,k}$ of task τ_i arrives at time $r_{i,k}$, the migration method utilizes Algorithm 1 to choose a runqueue $rq(\tau_i)$ for τ_i (that is, a core on which τ_i will be scheduled). The algorithm makes use of information about τ_i as well as the status of the multiple runqueues. As a result of this data, it attempts to schedule activities in such a way that the runqueues are not overcrowded $(\forall j, Uj \leq 1)$ while lowering the number of migrations. If $U_{rq(\tau_i)} \leq 1$ (Line 1), $rq(\tau_i)$ remains unaltered, and the job is not moved (Line 2). Otherwise (lines 4–18), a suitable runqueue $rq(\tau_i)$ is chosen as follows:

- If $\exists j : U_j + C_i/P_i \leq 1$, then choose the first runqueue j that has this property: $j = min\{h : U_h + C_i/P_i \leq 1\}$ (Lines 4 — 8). Lines 4 — 8 implement the well-known First-Fit (FF) heuristic, although alternative heuristics such as Best-Fit (BF) or Worst-Fit (WF) can also be utilized here.



```
Data: Task $\tau_i$ to be placed with its current
      absolute deadline being $d_{i,k}$; state of all the
      runqueues (overall utilisation $U_j$ and
      deadline of the currently scheduled task $d^j$
      for each core $j$)
Result: rq($\tau_i$)
1  if $U_{rq(\tau_i)} \leq 1$ then
       /* Stay on current core if schedulable */
2      return $rq(\tau_i)$
3  else
       /* Search a core where the task fits   */
4      for $j = 0$ to $M - 1$ do /* Iterate over all
        the runqueues                          */
5          if $U_j + C_i/P_i \leq 1$ then
6              return $j$ /* First-fit heuristic    */
7          end
8      end
       /* Find the runqueue executing the task
          with the farthest away deadline      */
9      h = 0
10     for $j = 1$ to $M - 1$ do /* Iterate over all
        the runqueueus                         */
11         if $d^j > d^h$ then
12             h = $j$
13         end
14     end
15     if $d^h > d_{i,k}$ then
           /* $\tau_i$ is migrated to runqueue $h$, where
              it will be the earliest deadline
              one                              */
16         return $h$
17     end
       /* Stay on current runqueue otherwise  */
18     return $rq(\tau_i)$
19 end
```

Algorithm 1: Algorithm to select a runqueue for a task τ_i on each job arrival.

Fig. 4. adaptive partitioned pseudo code

[3]

- Once the execution flow reaches reaches Line 9, this indicates that $\forall j, U_j + C_i/P_i > 1$ (task τ_i does not fit on any runqueue). Then, like the gEDF methodology does (Lines 9 — 17), choose a runqueue j by comparing the absolute deadlines $d_{i,k}$, and $\{d^j\}$:
  - If $d^h \equiv max_j\{d^j\} > d_{i,k}$ (as found in Line 15), then pick the runqueue h that is currently processing the task with the most distant deadline in the future (Line 16).
  - Otherwise, the task should not be migrated (as shown in line 18).

A quick observation shows that Lines 9 — 14 demonstrate how to compute $max_j\{d^j\}$ by instantiating on all the runqueues, however the Linux kernel keeps all the d_j in a heap, thus the maximum may be reached with a logarithmic complexity in the number of cores. The theorems and proofs highlighted below have been taken from the paper by Abeni and Cucinotta [3]

Theorem 1. *The apEDF method can schedule any taskset with $U = \sum_i C_i/P_i \leq (M+1)/2$ without compromising it's deadline. [3]*

Proof. Because $rq(\tau_i)$ is set to 0 when τ_i is formed and is only updated when $U_0 > 1$ (thus, the check at Line 1 fails) using the First Fitheuristic (Lines 4 — 8), it can be shown that if First Fit heuristics can generate a schedulable task partitioning, then Algorithm 1 behaves and has the First Fit characteristics. Previous work [8] demonstrated that if $U \leq (M+1)/2$, then FF provides a schedulable partitioning, implying that apEDF can appropriately schedule tasksets with $U \leq (M+1)/2$ without missing any deadlines. [3] □

Theorem 2. *If apEDF is able to discover a scheduling partitioning, the migrations will halt once it is found. [3]*

Proof. The check in Line 1 of the algorithm pseudo code guarantees that if task τ_i has already been entered in a runqueue with $U_i \leq 1$, it is no longer migrated; hence, only jobs allocated to overcrowded cores (possibly suffering from missed deadlines) are migrated. As a consequence, tasks can initially migrate, but if Algorithm 1 finds a schedulable partitioning, the tasks no longer migrate. □

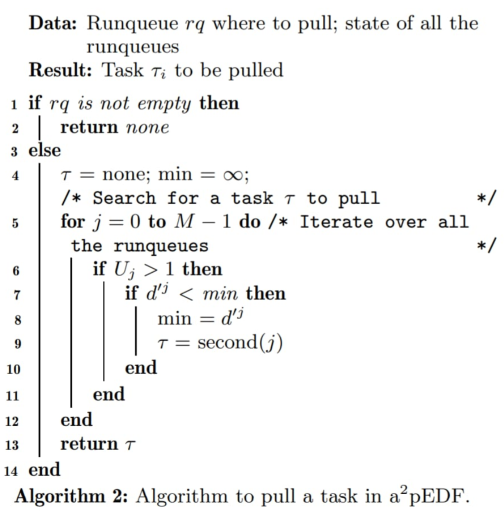
Theorem 2 demonstrates a significant difference between apEDF and r-EDF. At each task deadline $d_{i,k}$, the latter "forgets" the core on which a task has been performed, lowering U_j by C_i/P_i at that moment and potentially migrating tasks at each job arrival/activation, even if they are appropriately partitioned. Instead, Algorithm 1 prevents unnecessary migrations by allowing tasks to remain on the same core as long as there are no overloads, adjusting the runqueues' utilizations only when tasks migrate (and not when they de-activate).

Theorem 1 asserts that if $U < (M+1)/2$, then apEDF can instantly establish a schedulable partitioning without overloading or migrations (that is, just one initial migration from core 0 is required). If the taskset's usage exceeds $(M+1)/2$, some extra migrations may be required. Theorem 2 then shows that if a schedulable partitioning is found after these initial migrations, no further migrations will occur: if the FF heuristic used by apEDF is unable to immediately find a schedulable partitioning, apEDF migrates a task at each job arrival until the schedulable partitioning is found.

Lines 9 — 17 of Algorithm 1 ensure that the M earliest-deadline tasks are either scheduled or placed on non-overloaded cores if a schedulable tasks partitioning does not exist. Inherently, this system attempts to ensure that tasks/jobs with short absolute deadlines are not starved and that the discrepancy between the present time and the absolute deadline is limited. As a result, it has been proposed that if $U \leq M$, then each task will still experience bounded tardiness ($\exists L : \forall \tau_i \in \tau, max_k\{f_{i,k} - d_{i,k} \leq L\}$) even if such a schedulable division does not exist. In other words, the apEDF is intended to have the benefits of both pEDF and gEDF.

B. Reducing Tardiness

For the reducing the tardiness of the adaptive partitioned scheduling, further modifications were done on algorithm 1. In contrast to apEDF, a^2pEDF does not require restricted migrations, because a "pull" operation can move a job after it has begun to run on a core (and has been interrupted by an earlier-deadline task). Line 1 of the algorithm above (pull tasks solely to idle cores) may appear weird at first glance.



Data: Runqueue rq where to pull; state of all the runqueues
Result: Task τ_i to be pulled

```
if $rq$ is not empty then
    return none
else
    $\tau$ = none; min = $\infty$;
    /* Search for a task $\tau$ to pull */
    for $j = 0$ to $M - 1$ do /* Iterate over all the runqueues */
        if $U_j > 1$ then
            if $d'^j < min$ then
                min = $d'^j$
                $\tau$ = second($j$)
            end
        end
    end
    return $\tau$
end
```

Algorithm 2: Algorithm to pull a task in a^{2}pEDF.

Fig. 5. Modified adaptive partitioned pseudo code

[3]

It is prompted by the fact that after a task completes, its usage may only be reused after the task's deadline (or, more precisely, after the so-called "0-lag time" [3]). As a result, instantly taking a task from a different core may result in a momentary overload and additional missed deadlines. In principle, a^2pEDF may avoid this issue by waiting for the 0-lag time (or the end of the task period) before executing a "pull request" operation, however doing a "pull request" instantly only when the core becomes idle allows for a simpler solution (without having to set up timers for pulling). Also,as stated in the paper by Abeni and Cucinotta [3]

Although apEDF can provide a bounded tardiness if $U \leq M$,the tardiness bound L can be quite large (much larger than the one provided by gEDF), as it will be shown in Section 5.This is due to the fact that the runqueue on which a job $J_{i,k}$ is enqueued is selected at time $r_{i,k}$ when the job arrives; if task τ_i does not fit on any runqueue, the target runqueue is selected based on the absolute deadlines $\{d^j\}$ of the jobs that are executing on all cores at time ri,k, so the selection can be sub-optimal after some of these jobs have finished.For example, if at time $r_{i,k}$ job $r_{J,k}$ is inserted in the j^{th} runqueue, then it will be scheduled on the j^{th} core even if some other core in the meanwhile becomes idle: in other words, apEDF is not work-conserving.

V. CONCLUSION

This paper presented an overview of the methodology of scheduling tasks on multi-cores using EDF. In particular, major focus was given to adaptive partitioning and how it contributed towards achieving an optimal solution in multi-core scheduling. By comparing it to traditional gEDF and pEDF techniques, we found that it outperformed gEDF in terms of hard real-time performances and also better than pEDF in terms of soft real-time performance. However, in the paper by Abeni and Cucinotta [3], a comprehensive collection of simulations was used to compare the hard and soft real-time performance of the two algorithms, with the finding that the a^2pEDF algorithm delivers less latency than the apEDF algorithm when utilization is high (and the number of tasks is small). In this case, apEDF performs somewhat worse than gEDF, however a^2pEDF still performs better than gEDF. Further discussions regarding implementing adaptive migrations, Dynamic Task Arrivals and Terminations have been detailed in the above mentioned paper that served as a major source towards this research.

REFERENCES

[1] T. Gleixner and D. Niehaus. Hrtimers and beyond: Transforming the Linux time subsystems. In proc. of the Linux Symposium (OLS'06), pages 333–346, Ottawa, Ontario, Canada, 2006.
[2] Real-Time Scheduling on Multi-core: Theory and Practice BY Paulo Manuel Baltarejo de Sousa
[3] EDF Scheduling of Real-Time Tasks on Multiple Cores: Adaptive Partitioning vs. Global Scheduling by Luca Abeni and Tommaso Cucinotta
[4] A. Bastoni, B. B. Brandenburg and J. H. Anderson, "An Empirical Comparison of Global, Partitioned, and Clustered Multiprocessor EDF Schedulers," 2010 31st IEEE Real-Time Systems Symposium, 2010, pp. 14-24, doi: 10.1109/RTSS.2010.23
[5] Sanjoy Baruah and John Carpenter. 2003. Multiprocessor Fixed-Priority Scheduling with Restricted Interprocessor Migrations. In Proceedings of the 15th Euromicro Conference on Real-Time Systems (ECRTS 2003). IEEE, Porto, Portugal, 195–202.
[6] Simulation of Multi-core Scheduling in Real-Time Embedded Systems Md. Golam Hafiz Khan
[7] J. M. L´opez, M. Garc´ıa, J. L. Diaz, and D. F. Garcia. Worst-case utilization bound for EDF scheduling onreal-time multiprocessor systems. In Proceedings of the 12th Euromicro Conference on Real-Time Systems (ECRTS 2000), pages 25–33, Stockholm, Sweden, June 2000. IEEE.
[8] An Experimental Comparison of Different Real-Time Schedulers on Multicore Systems by Juri Lelli, Dario Faggioli, Tommaso Cucinotta, Giuseppe Lipari
[9] N. Saranya and R. C. Hansdah, "Dynamic Partitioning Based Scheduling of Real-Time Tasks in Multicore Processors," 2015 IEEE 18th International Symposium on Real-Time Distributed Computing, 2015, pp. 190-197, doi: 10.1109/ISORC.2015.23.
[10] An EDF-based Scheduling Algorithm for Multiprocessor Soft Real-Time Systems James H. Anderson, Vasile Bud, and UmaMaheswari C. Devi
[11] A Comparison of Global and Partitioned EDF Schedulability Tests for Multiprocessors (TR-051101) Theodore P. Baker
[12] A. Bastoni, B. B. Brandenburg and J. H. Anderson, "An Empirical Comparison of Global, Partitioned, and Clustered Multiprocessor EDF Schedulers," 2010 31st IEEE Real-Time Systems Symposium, 2010, pp. 14-24, doi: 10.1109/RTSS.2010.23.